SDRL Q14-02021-B

Q14 - Standards Development Plan

Ada Interfaces to SQL: Analysis and Recommendations

Prepared for
Software Technology for Adaptable
Reliable Systems (STARS)
Unisys STARSCenter
Shipboard and Ground Systems Group
Reston, Virginia

Prepared by
Unisys Corporation
Defense Systems
System Development Group
P.O. Box 517, Paoli, PA 19301-0517

Contract No. F19628-88-D-0031
IDWO P.O. 010412

March 20, 1989

[illegible]

1 Executive Summary

This report provides a description of the SQL Ada Module Extension approach to providing an interface from Ada to SQL. The SAME is recommended to STARS as the best available means of providing this interface. The SAME is an extension of the SQL module approach ~~[Boy87]~~ which is based upon the premise that Ada and SQL are separate languages, and should be programmed in separate source streams and compiled in their native environments. The interface between Ada applications and SQL queries is just that: an abstract interface, rather than the more murky interfaces presented by the popular embedded approaches. Perhaps the single most compelling advantage of the module approach is that it neatly bypasses many of the conceptual difficulties created by source level interweaving of languages (Ada and SQL) of significantly different typing and computational models.

At this time a working SAME support library is available, and has been used in Q14 to develop Ada–SQL applications using an INGRES relational DBMS. There is reason to believe that the SAME could be used today in the development of Ada applications requiring access to SQL databases. Further, the SEI is currently developing a SAME compiler which will automate the construction of the Ada–SQL interfaces which characterize the module approach.

Also included in this report is a detailed description of the criteria by which the SAME approach was compared with the WIS Ada/SQL approach. A detailed discussion of this critical comparison is included.

[illegible]

2 Introduction

In 1987 the Software Engineering Institute was directed to study the issue of Ada interfaces to SQL, and to propose a solution to this difficult problem. The first step in this process was a report [EFGW87] which outlined the characteristics of a "good" interface from Ada to SQL. An objective evaluation of proposed solutions to the problem of Ada interfaces to SQL [X3H88] [BFHH87] [Boy87] yields the conclusion that the SQL Module alternative is most promising.

Section 3 of this report provides a high-level overview of SAME, and introduces some of the terminology used to describe the SAME. A more thorough exposition can be found in [Gra88]. Section 4 presents a detailed description of the key characteristics of the SAME which influenced the Q14 decision to recommend the SAME to STARS. Section 5 presents an extended discussion of a criteria which can be used to evaluate alternative approaches to providing Ada interfaces to SQL services; also included in section 5 is a point by point comparison of the SAME to the WIS [BFHH87] Ada bindings to SQL, called Ada/SQL. Finally, section 6 presents the summary conclusions of this report.

3 Overview of SAME

The central idea of the module approach lies in the treatment of Ada and SQL as equally important, separate languages. That is, SQL is neither subsumed by, or embedded into, Ada source code. Instead, the module approach stipulates that SQL programming be done in SQL, and Ada programming be done in Ada. The interface between SQL and Ada code is achieved through use of a *module compiler*. The module compiler translates standard SQL module language into Ada source code, or object code which can be combined with Ada at link time.

SQL modules consist of a sequence of SQL statements; in [X3H86] only data manipulation statements (DML) may appear in SQL modules. The individual SQL statements define, e.g., SQL queries. A module compiler will generate an Ada interface to each such SQL query. The Ada interface to the SQL query will provide input to SQL as procedure input parameters, and return the results of the query to Ada as output parameters. The implementation of this interface, also generated by the module compiler, could be a *pragma interface* to the code generated by the DBMS SQL compiler.

One weakness of the module approach described above is that the Ada application interface to the SQL statement will be presented in terms of SQL-compatible types. Although at some point all interfaces between Ada and SQL will have to map application types to the underlying SQL DBMS supported types, it is not desirable to impose the relatively lax SQL type model on Ada applications. The direct interface between Ada and the SQL queries compiled by the SQL compiler is called the *concrete interface* precisely because it presents in the interface the concrete SQL types, i.e., the types of the relation's columns.

The SAME extends this model by placing an additional layer between the Ada application and the concrete interface, called the *abstract interface*. This interface presents a strongly-typed interface, with application-defined *abstract types* as parameters. The difference between abstract types and concrete types is best illustrated by example. In a database, the column types for an employee relation's employee-name and employee-city may well be represented by a simple character string, perhaps even strings of the same length. In SQL, there would be nothing illegal about comparing values of one column with those of the other. In Ada, the application would prefer to model employee-name and employee-city as distinct abstract types.

The application types in the abstract interface are called *domain types*, because they represent the abstract domains which correspond to the concrete SQL domains ("columns"). These types are *derived* from types provided in the SAME support library. The support library defines a set of abstract data types which encapsulate the representation and semantics of the needed Ada–SQL type mapping. Thus, SAME provides a measure of controlled Ada–SQL type extensibility. If Ada types (e.g., enumeration types, decimal types) are to be represented in the relational database, an enumeration support ADT will be written and inserted into the SAME support library (NOTE: these ADTs are already defined and implemented in the SAME).

4 Characteristics of the SAME Approach

The SAME layered interface approach and abstract domain typing approach provides opportunities to introduce other desirable Ada characteristics into the Ada–SQL interface, such as robust error detection and safe treatment of SQL null values. The following subsections outline these salient characteristics of the SAME. It is hoped that the sum of this description will provide the reader with some measure of the elegance as well as problems inherent in the SAME. However, by far the best description of the nuances of the SAME can be found in [Gra88], which includes example SAME programs in addition to numerous discussions of design rationale for the SAME approach.

4.1 Finite 'State Modifier' Parameterization

Marc Graham (SEI) has pointed out an interesting distinction between the Ada interface problem posed by SQL and, say, POSIX or the X Window System. Essentially, systems such as POSIX, X, and DIANA can be characterized by a finite set of state-manipulation interfaces. SQL, however, is a programming language, and no such finite number of state modifier functions are definable. Thus, *inherent in any interface of Ada to SQL is some element of interface generation.*

The Ada/SQL approach generates operations from an Ada schema specification, resulting in enormous compile-time resource consumption and the need for some kind of pre-compiler to filter overloaded operations not actually used by the DB application program. Embedded SQL uses pre-compilation to map embedded SQL statements to a set of interfaces which do not so much define an SQL interface as they do provide generalized run-time services to a backend DBMS.

In the SAME, this interface generation is elevated to the status of principle design activity, rather than by-product (or means to an end). Each SQL query is viewed from the Ada application as a call to an abstract interface defined for some SQL services. The details by which this service is rendered is transparent to the application, which is freed from concerns such as mapping SQL data types to their abstract types in the Ada program, from concerns about database error conditions, and freed from the implementation details by which the query was satisfied.

The abstract interface constitutes a kind of *external schema* [Cle85] which provides the Ada application program a high degree of data independence. For example, the application program would be well insulated from schema reorganization (schema evolution), which would in most cases require only relinking of the application rather than any recoding. In contrast, any Ada–SQL interface solution which is achieved by co-mingling Ada and SQL source is fundamentally weaker in this regard. Of course, with proper design discipline some degree of data independence can be achieved in embedded approaches. None of the embedded approaches mandate definition of abstract interfaces, as is the case in the SAME.

4.2 Separating Distinct Language Paradigms

Ada and SQL are distinct languages, each with distinct typing and computational models. Ada is a procedural language, with a notion of "sequence". In contrast, SQL is a query specification language which has a more declarative rather than procedural reading. Ada is also a strongly typed language, with a notion of type extensibility via data abstraction and type constructors. SQL is a weakly typed language. These and other distinctions create conceptual and pragmatic barriers to interfacing SQL and Ada.

Consider some implications of the distinct type models of SQL and Ada. Note first that in any interface between Ada and SQL, queries will eventually be processed by SQL using the rules of SQL. Thus it is not possible to impose the Ada type model on SQL databases. Rather the interface must clearly specify a type mapping to and from Ada and SQL types.

For example, SQL has a notion of "string" type, as does Ada (there are other examples, e.g., distinctions between arithmetic types such as SQL Float and Ada fixed-point numbers, but the string example should suffice). However, the semantics of string processing in SQL is subtlely distinct from the semantics of Ada strings. SQL defines rules for string comparison which would allow strings of unequal length to nevertheless be "equal" (SQL defines "padding" semantics for such comparisons). In Ada such strings would never be equal. The Ada–SQL interface should make such distinctions clear. In Ada/SQL, the semantics of strings at the interface are those of SQL; however, string values returned by Ada/SQL *target specifications*, e.g., fetch...into statements, are converted to native Ada string objects, with Ada string semantics. Thus the application code treats strings with subtley different semantics than does the DBMS.

In the SAME, the goal is to provide a clear separation of SQL and Ada type semantics by processing each language under its own native environment; the type mapping semantics are then projected onto an interface (see next section) which encapsulates the type mapping semantics. In the SAME, this interface is defined by the SAME support library, and the domain types derived from this library. The type semantics encapsulated in the SAME library support a uniform view of application types, both within application and the DBMS.

4.3 Abstract Domain Types and Concrete SQL Types

In the SAME, the mapping of SQL types to Ada types is accomplished via the definition of abstract domain types (in Ada). A SAME domain type consists, in general, of two derived type definitions and one generic package instantiation. The database application program deals with domain types, not SQL types.

The underlying types from which the domain types are *derived* are defined in the SAME support library. These support types encapsulate the SAME mappings from the concrete representation of types in the underlying database (as defined in the SQL standard [X3H86]) to their corresponding types in Ada. Thus the application programmer is two levels of abstraction away from the DBMS-specific encodings of the standard SQL types. This distance, as well as the architectural localization of this type mapping, simplifies the task of hosting SAME on various DBMS platforms, and insulates the application program from subtle changes in the SQL standard types (because of the encapsulation of the support types as abstract data types).

Although this type mapping scheme introduces additional complexity for application programmers, there are a number of benefits in addition to those listed above, including safe treatment of SQL null values, robust error handling, and improved database application design. These points are discussed in the following three subsections.

4.4 Safe Treatment of SQL Null Values

One significant difference between Ada's and SQL's notion of type *values* is the SQL *null value*. The null value represents the value "unknown", and introduces a notion of incomplete information alien to Ada, where a type is defined as a set of values and a set of operations; there is no Ada value which represents the unknown value. In SQL, the occurrence of a null value is indicated by an *indicator* parameter to a query. It is the responsibility of the invoker of a query to evaluate the indicator parameter and take appropriate action in the case of a null value.

Mistreatment of SQL null values can lead to programming errors that are obscure and difficult to diagnose. The SAME approach guarantees safe handling of null values by freeing the application programmer from his obligation to check the indicator parameter. Instead, the programmer declares his intent to allow or disallow null values through use of the appropriate domain type. Note that the SAME distinguishes the application programmer from the interface programmer. Although the interface programmer is *not* freed from the responsibility of correctly handling SQL null values, as discussed later in this report the development of a SAME compiler will automate interface programming.

It was noted above that domain types consist of two type derivations and a package instantiation; the two base derived types correspond to null bearing and non-null bearing types. The SAME is guaranteed to raise an exception if a non-null bearing type receives a null value; further, the SAME guarantees consistent handling of null values, as null valued semantics is encapsulated in the support types abstract data type definitions.

4.5 Robust Error Handling

A corollary to the above discussion on safe treatment of null values is the generalized notion of robust error detection and recovery. Again, the interface problem stems from differences in Ada and SQL, in this case from distinct interpretation of error conditions. In SQL the notion of "error" is generalized into a single integer code (SQLCODE) which represents an indicator for the current state of the underlying DBMS. A few values of SQLCODE are defined by the standard (e.g., 0 means a successful "fetch", 100 means end of file), but most – including the real error states (defined as negative SQLCODE values) are left to the DBMS vendor to define.

The onus of detecting, evaluating, and acting upon the value of SQLCODE is placed upon the application developer. This presents at least two problems. First, there is no assurance that the application program will correctly recognize erroneous database conditions – indeed, there is nothing to mandate that the application program will examine the SQLCODE. This lack of robustness stands in sharp contrast to the goals of Ada software development. Second, the encoding of vendor-specific SQLCODE values into software is a clear obstacle to application portability, in this case portability across different DBMS vendors.

In the SAME, one task of the application and SAME interface developer is to specify which SQLCODE values represent meaningful results for a particular SQL request; all values not in this set will invoke the SAME error processing system, which will eventually result in the raising of an exception. Such values usually correspond to an error situation which the application program could not recover from in any event. Note: the SAME error processing facilities do provide escape mechanisms which would allow the application to make some attempt at recovery.

One immediate result is that the application developer is not aware of the notion of SQLCODE, but can instead code his application under the assumption that all requests to the underlying DBMS (through SQL) have completed successfully and without error. Furthermore, both application and interface code are clearer. Application code is clearer since it can assume error-free processing. Interface code is more self-documented since it is more informative to enumerate the few valid SQLCODE values than write code to process all possible invalid codes.

4.6 SAME Support for Sound Database Application Design Principles

Perhaps no single defining characteristic so clearly delineates SAME from embedded approachs to interfacing SQL as the way in which SAME supports – mandates, actually – a sound database application design approach. As already mentioned, interfacing Ada to SQL requires at some point the generation of application-specific interfaces to the SQL processor; in the SAME, these interfaces become a focal point of the application design process, rather than a secondary artifact.

In practice, this means that the interface from an application to the DBMS are defined early in the application development process; using the SAME makes it difficult to formulate the application – DBMS interface in an *ad hoc* fashion. This is clearly a desirable result for several reasons.

Defining the required DBMS services is an integral part of the design of any database application. Embedded approaches in effect grant the application coder the privilege of making use of the DBMS in an uncontrolled and perhaps inappropriate fashion (e.g., creating but forgetting to "drop" temporary tables). In cases where the designer and coder are the same person, this may be tolerable (although arguably still an inappropriate process to develop software); however, in larger organizations where application coders may not be as experienced as the system designer, such a situation may be less pardonable.

Early definition of the abstract interfaces to SQL services also requires a clear and documented mapping of SQL database types to the abstract types to be manipulated by the application programmer. Interestingly, in traditional database application design, the database designer will go through a process of schema definition which begins with identification of the abstract objects which the database applications will manipulate. These abstract objects are then represented in the underlying database using types supported by the DBMS. Unfortunately, the abstract objects defined during database design are not captured in the schema, and this important information is often lost. In environments where only Ada will be operating on the DBMS, or where an Ada application will be the originator of a particular schema, the SAME provides a vehicle for capturing and making use of this information. In cases where existing schemas will be used by SAME applications, this information may be harder to derive.

4.7 SAME Complexity: Need for Automation, and Approaches

The SAME derives much of its effectiveness through explicit handling of critical Ada – SQL interface issues (e.g., type mapping, error handing, type extensibility, null values). However, the SAME is also overtly complex; although the application programs remain clear, the implementation of the abstract and concrete modules remains a tedious and complex task. This is perhaps the weightiest criticism which can be leveled at the SAME (although in fairness it can also be argued that all problems can be characterized by some minimum level of complexity which will be exhibited by their solutions).

It is clear that interface programming in the SAME is tedious, and requires some means of automating the definition and implementation of the SAME interfaces (domain packages, abstract and concrete modules). Figure 1 illustrates a fragment of an abstract module for the SAME.

Fortunately, SAME tools are in design, and prototypes are expected to be available by August 1, 1989. The following subsections describe one possible SAME tool configuration, and issues raised by SAME tooling.

```
procedure FETCH(TUPLE : in out PART_NBR_CITY_PAIRS; FOUND : out boolean) is

   CITY_BUF: CHAR (1 .. 15);
   CITY_IND: INDICATOR_TYPE;

begin
   CONCRETE_MODULE.FETCH(TUPLE.PNO, CITY_BUF, CITY_IND, SCP.SQLCODE);
   case SCP.SQLCODE is
      when NOT_FOUND =>
         FOUND := false;
      when SQL_ERROR =>
         SDEP.PROCESS_DATABASE_ERROR;
         raise SCP.SQL_DATABASE_ERROR;
      when others =>
         if CITY_IND < 0 then -- null city found
            ASSIGN(TUPLE.CITY, NULL_SQL_CHAR);
         else
            ASSIGN(TUPLE.CITY,
                   CITY_OPS.WITH_NULL(CITY_NOT_NULL(CITY_BUF)));
         end if;
         FOUND := true;
   end case;
end FETCH;
```

Figure 1: SAME Abstract Module Code Fragment

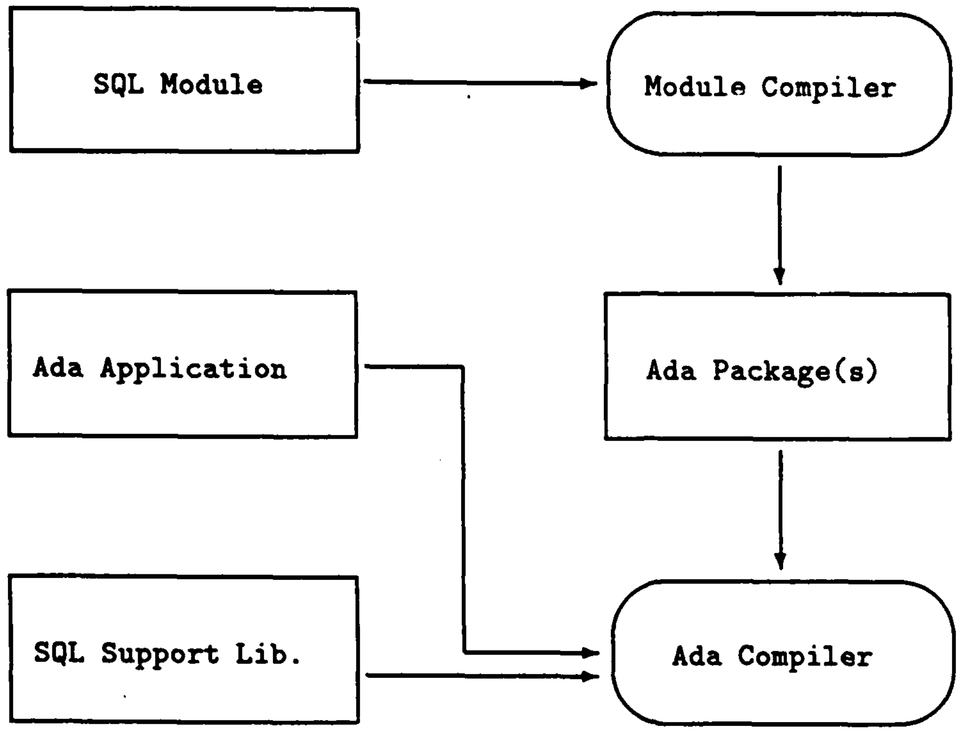

Figure 2: SQL Module Approach

4.7.1 Module Compilers and SAME Compilers

The SAME, as it's acronym suggests, is an extension of the SQL module approach to support more fully the Ada language. Module compilers for the module approach – not SAME – have been implemented, e.g., XDB on UNIX Unisys 5000 and MS-DOS personal computers, and Datacomm, MVS/CICS for the Army SIDPERS-3 project. These systems work by compiling standard SQL module language into Ada code which accesses the DBMS. In SAME terms, these module compilers generate *concrete modules*, so named because the interface to this code projects the concrete type model employed by the SQL processor.

The key to the SAME automation is to extend the SQL module language, either through bona fide language extensions or comment-style annotations, with the additional information required to generate the abstract domain types, abstract modules, and other ancillary concerns such as packaging. Thus the solution which presents itself is a SAME compiler. Figures 2 and 3 illustrate the distinction between the vanilla module compilers and the SAME module compiler.

NOTE: these figures do not illustrate dependencies among generated Ada code (i.e., "with"s), but merely identifies generated vs. handwritten Ada source.

4.7.2 Third Party Solution: Syntactic Dialects

One problem with the SQL standard is that it is defined as an intersection of vendor features; thus the situation that the standard defines a language which no one supports but many comply with

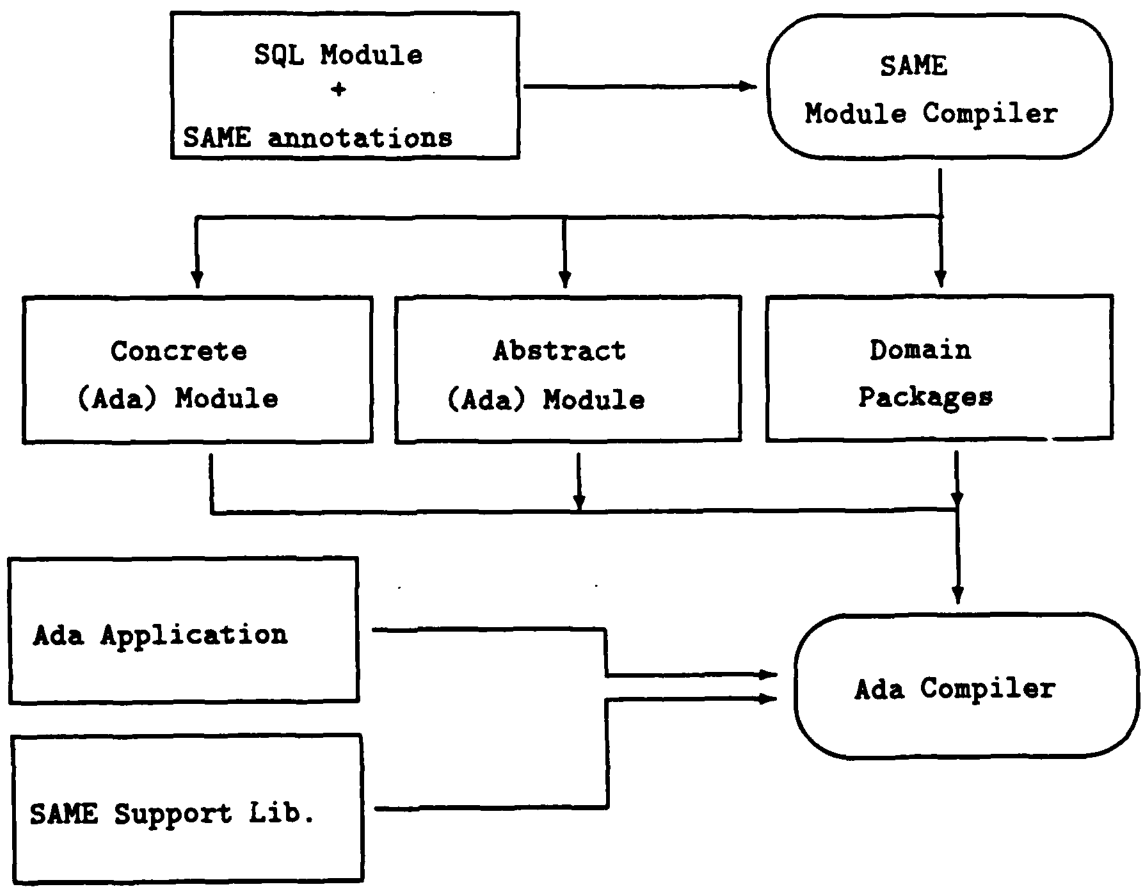

Figure 3: Possible SAME Tool Approach

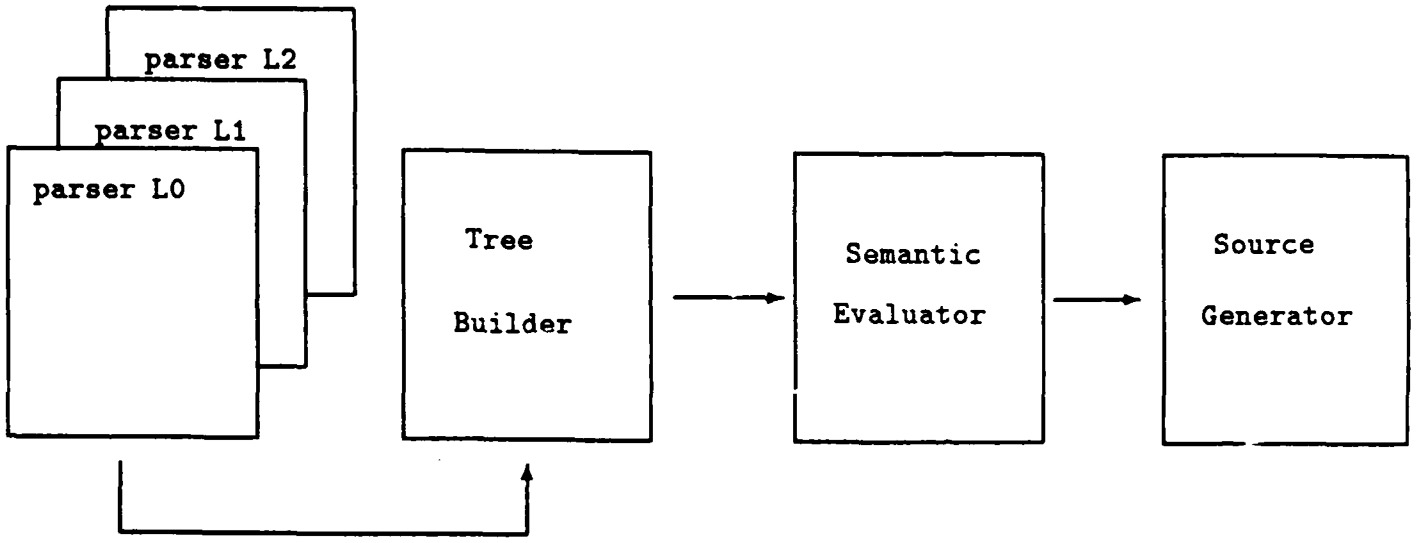

Figure 4: SAME Compiler Families by Generation

[Dat87]! If the SAME is to succeed, in the sense that it can be adopted by Ada developers to a wide range of target DBMS, the SAME compiler must be flexible enough to support multiple vendor implementations of SQL.

As suggested by the classification of the SQL standard as an "intersection" standard, most DBMS vendors support a common core set of SQL features but then vary with vendor specific features (e.g., dynamic DDL statements, database connect statements). This raises a dilemma: if we are to promulgate a SAME compiler for widespread use in STARS, we must be able to construct such a tool without having to rely on DBMS vendors. Yet, how can such a tool be constructed economically if each one is by definition vendor-specific (based upon the vendor's SQL dialect)?

The solution is to devise a SAME compiler which is constructed with the presumption of supporting several SQL *syntactic dialects*. As the term implies, a syntactic dialect L' of language L conveys the same semantic content, albeit structured in a slightly (syntactically) different form. If this goal is achievable, then the SAME compiler can be specified and implemented with respect to a common intermediate representation for the SQL module language, with syntactic variants mapped to the common intermediate form at parse time.

The approach to achieving this objective which is currently under consideration by Marc Graham is the use of the Unisys generation system, SSAGS [PKP*82]. SSAGS is in essence a powerful compiler-compiler which generates parsers, tree-builders, semantic evaluators, and code (source) generators from a high-level specification language. A direct benefit of using SSACS is its built-in support for the separation of *concrete grammars* (the actual SQL dialects) and *abstract grammars* (the intermediate representation). The SSAGS generated semantic analyzer and code generators process the output abstract grammar, and so are insensitive to the possibility of multiple input concrete grammars.

Figure 4 is an abstract illustration of a SSAGS-generated family of SAME module compilers.

4.7.3 The SAME Module Language

The question arises: "What should the SAME input language *look like*?" There are two basic positions, and a compromise position. T' SAME design committee has not committed to a particular position, and all have been and are under evaluation. Essentially, the three positions are:

- Standard SQL module language with comment-form annotations
- A concise SAME language
- Extended SQL module language

The first approach extends the standard module language transparently via comment-style annotations, similar to the way Byron and Anna extend Ada. This idea is based upon the observation that the module language contains sufficient information to generate the concrete module; the annotations would provide additional information concerning the Ada–SQL type mapping, recognized error conditions, domain types packaging, etc. Figure 5 illustrates a fragment of extended module language.

There are some advantages to this language approach. First, preservation of the standard module language means several languages (or systems) can share the same SQL mc·lule specification; since the annotations are transparent to existing module compilers, they can be ignored. Also, considerable effort has been expended in defining the SQL module language. Preserving the language and merely adding syntax for the SAME builds upon this considerable language design investment.

Unfortunately, it has been observed that the comment extensions to the module language results in a cumbersome language for the SAME, with some degree of redundancy. This aesthetic problem could pose a serious obstacle to widespread acceptance of the SAME compiler.

An alternative approach to the SAME language would be to define a new language "from scratch", such that the language would contain all the necessary information to generate a corresponding standard module language specification in addition to the Ada particulars. Such a language would not be constrained by the SQL module language syntax, but could, for example, be based upon different language principles (e.g., pure relational algebra, or perhaps SQL including the modifications mentioned in [Dat87]).

Perhaps the most obvious argument against this position concerns our desire to see a SAME processor prototyped by August 1989; it would seem that designing a new language would require considerable effort, besides the obvious risks to industry acceptance. Such an approach, while feasible in a pure research setting, would not provide a satisfactory near-term solution.

The final alternative, simply extending the SQL module language in order to facilitate SAME compiler derivation of much of the information couched in the "-&" annotations, is currently under investigation. Since no complete concrete syntax has been proposed, it is difficult to evaluate the aesthetics of this proposal. The obvious advantage is a more convenient syntax for SAME users. The disadvantages are additional configuration management problems in multi lingual environments (i.e., the SAME module would not be accessible to these tools, although perhaps pure SQL module language output from SAME would be accessible; this introduces configuration management issues). Further, there are some risks to industry acceptance due to variance from an existing standard, although the risks appear to be smaller than in the "from scratch" language alternative.

```
-- SQL module comments are introduced by the ''--'' characters
-- SAME annotations are represented by the ''&'' suffix on SQL module comments

--& domain PNO is new SQL_CHAR (5);
--& domain SNAME is new SQL_CHAR (20);
--& domain CITY is new SQL_CHAR (15);
--& domain STATUS is new SQL_INT range 0 .. SQL_INT'range;

create table S (
   SNO    CHAR (5) NOT NULL, --& domain : SNO
   SNAME  CHAR (20),         --& domain : SNAME
   STATUS INT,               --& domain : STATUS
   CITY CHAR (5),            --& domain : CITY
   UNIQUE ( SNO ) )

-- other table and domain definitions here ...
```

Figure 5: Sample SQL Module Language Annotations

4.8 Summary of SAME

The SAME addresses the most difficult issues raised by attempting to interface two languages of radically different paradigms. The solution offered by the SAME is flexible, makes good use of Ada and SQL language capabilities, and is designed to impose a minimal interface overhead. On the minus side, SAME applications, or rather the application specific SQL interfaces, are quite complex, and are tedious to construct. Fortunately, most of this tedium is automatable; the SAME compiler under design at this time will accept SQL module language (some variant) and automatically generate the abstract and concrete modules and domain packages. This tool should make the SAME the Ada–SQL interface of choice for production quality Ada SQL applications.

5 A Comparative Analysis: SAME vs. Ada/SQL

A substantial investment has been made in the WIS Ada/SQL bindings, and therefore no discussion of Ada–SQL bindings can be complete without consideration of this important alternative. Other alternatives to embedding SQL in Ada in comment-form are not as noteworthy since they unnaturally extend Ada by preprocessor, whereas Ada/SQL uses the inherent extensibility provided by Ada.

The Lockheed SQL work performed under a STARS Foundations contract is not sufficiently distinct from the WIS Ada/SQL work to warrant a separate evaluation. It is mainly distinguished from Ada/SQL by the approach to generating the overloaded domain operations, which is accomplished via successive instantiation of a nested generic data dictionary specification. In fact, this generation technique was the main thrust of the Lockheed work. All of the points in the following evaluation apply equally to Ada/SQL and the Lockheed prototype.

At the outset it must be stated that, from the SQL programmer's perspective, Ada/SQL provides an elegant interface to SQL. Given a strict requirement that all aspects of an Ada–SQL interface must be coded in Ada, it is difficult to conceive a better solution. One key presupposition in the SAME is that this requirement is wholly inappropriate for this particular interface problem; SQL and Ada, as fundamentally distinct entities, should be treated as such, and SQL should not be subsumed or embedded in Ada.

5.1 The Evaluation Criteria

In order to establish an objective comparison, it is necessary to formulate criteria, a set of features, by which systems are compared. Table 1 provides an overview summary of one such set. Many, but not all, of the criterion were derived from [EFGW87]. In the following evaluation, several conclusions are suffixed with a "?" to indicate that the author has less certainty concerning the conclusion, or that there is room for debate concerning the conclusion. In these cases an attempt is made to present Ada/SQL counter-arguments in an honest way.

One point of note is that the evaluation which follows is based upon current implementations of the Ada/SQL model. [BF88] indicates that implementations compliant with the Ada/SQL specifications could generate, from application source, embedded SQL for a commercial preprocessor, calls to external SQL modules as in the module approach, or any host of alternatives. From this perspective, one could view the Ada/SQL approach as a solution which is symmetric to the automated SAME – rather than generate the interfaces from standard module language, generate the interfaces from Ada source. However, the author knows of no such implementations, and the evaluations below apply to current implementations only.

5.2 Applying the Criteria

5.2.1 User Code Portability

User code portability must take into consideration portability across hardware platforms, Ada compilers, and DBMS. It must be assumed that Ada code is written in a machine independent way, and so applications using the SAME and Ada/SQL should achieve a high degree of application code portability.

This evaluation is somewhat muddied in the SAME because there are really three classes of application software: the application itself, the SAME interfaces, and the SQL code. There's also a distinction between the automated SAME (i.e., with SAME compiler) and the manual SAME. In any event, application code uses an abstract interface, and so will be portable.

In the manual SAME, some additional portability issues are raised since each concrete module implementation will need to be examined for e.g., pragma interface syntax, which is not fixed by Ada. Also, the manual SAME might require hand modification of the SQL code to comply with vendor idiosyncrasies. Thus, manual SAME appears to present more obvious portability problems than Ada/SQL.

Criteria	SAME	Ada/SQL
User code portability	yes	yes
Third party solutions	yes	yes(?)
Pure Ada solution	no	yes
Standard preservation	yes	no(?)
Performance - Compile time	yes	no
Performance - Run time	yes	no
Ease of use	no	yes
Support DB design methods	yes	no
Effective use of Ada	yes	yes
Effective use of SQL	yes	no
Well defined type mapping	yes	yes(?)
Data independance, schema evolution	yes	no
Data interoperability to non-Ada applications	yes	no
Ease of validation	yes(?)	no(?)

Table 1: Ada-SQL Interface Evaluation Criteria

In the automated SAME, the application program is still portable, and further the SAME interfaces and concrete modules are *generated* from the SAME language. Thus the issue of SAME portability reduces to the identical constraints imposed on Ada/SQL portability, i.e., portability of the *system* to a host DBMS and Ada compiler.

On the other hand, it must be recognized that Ada/SQL is not isolated from potential portability problems raised by vendor SQL idiosyncrasies. It is an open question whether this could result in more insidious forms of portability difficulty: it is not clear that problems arising from vendor "quirks" can be isolated and easily resolved, since SQL statements appear passim in the application code. One example of such difficulties concerns character set independence. Not all DBMS process strings in ASCII, yet Ada/SQL interfaces manipulate Ada strings, which are by definition ASCII. Further, the Ada/SQL implementation constructs ASCII string representation of SQL queries to be processed by the SQL processor; a non-ASCII DBMS will not accept such strings.

5.2.2 Third Party Solutions

This is really an issue of practicality: is it likely that third party commercial vendors will make available production quality versions of either the SAME or Ada/SQL? Two potential classes of problems could prevent this desirable result: either method could, in practice, require access to either compiler or DBMS internals in order to produce efficient application code, or else the implementation cost of either method could be prohibitively expensive.

It is clear that in this category, the SAME has a marked advantage. This is because the SAME makes use of existing SQL processors, and does not duplicate the effort of SQL syntactic and semantic processing within Ada. Further, the cross-product generation of column operations in Ada/SQL exceeds most (all?) compiler limits; therefore a pre-compiler capable of full Ada overload resolution is required in order to examine the application source to determine which of these column operations are actually needed (the rest are filtered). This is an expensive prospect, and although a public domain Ada front-end is available, it is not validated.

Although SAME systems can be constructed by third party vendors, it may be the case that DBMS vendors will have an edge in producing production quality SAME module compilers. Access to the underlying DBMS data dictionary could provide, if nothing else, better integration of the SAME compiler with pre-existing DBMS utilities.

5.2.3 Pure Ada Solution

There is no question here: Ada/SQL allows the application programmer to develop an application in pure Ada, whereas in applications using the SAME someone, perhaps not the application programmer (e.g., database administrator) will have to write SQL code. To Ada/SQL advocates, this is one of the most appealing characteristics of Ada/SQL.

On the other hand, the author questions whether this is a significant benefit, or perhaps the root cause of severe difficulties in making Ada/SQL systems "production quality." In any event, the need for Ada/SQL pre-compilation filtering and difficulties with run-time performance of Ada/SQL query execution indicate that the pure Ada solution has not been achieved without cost.

5.2.4 Preservation of Standards

The Q14 mandate concerned Ada interface standards, and therefore this criterion is particularly important to the author. SQL is a language which is undergoing a formal standardization process in ANSI; we can expect the language to evolve in time (SQL2 is currently a draft standard). Any solution to the Ada–SQL interface problem must recognize this reality, and thus the solution must be adaptable to changes in both Ada and SQL.

The SAME has a clear edge here because of the separation of the Ada and SQL text. The SAME application programs, and indeed the abstract and concrete modules are, for the most part, insensitive to minor changes to the SQL standard. On the other hand, Ada/SQL embeds SQL statements – albeit in Ada syntax – throughout the application program. Further, the considerably complex pre-compiler is also sensitive to the SQL standard. All embedded approaches prima facie make it more difficult and expensive to maintain compliance with future versions of the SQL standard.

One point important enough to repeat is that SAME application programs view database resources through an external schema; this means that the SAME application is not only relatively impervious to schema evolution, but is also relatively isolated from the underlying conceptual schema. That is, the SAME application program would probably not change considerably under changes to the SQL standard, or for that matter migration of the implementation of the abstract and concrete modules to an entirely different data model.

5.2.5 Performance – Compile Time

This evaluation refers to existing implementations of the SAME and Ada/SQL. The negative evaluation of Ada/SQL compile time performance refers to the required pre-compilation pass. This essentially doubles the required compilation time, and this could be expensive in large-scale applications. Further, the tight coupling between Ada/SQL application code and the Ada/SQL data dictionary (which controls the generation of column operations) adds additional compilation (not link) dependencies.

On the other hand, as has been mentioned, SAME application programs are fairly impervious to modifications to the schema which do not "lose" information required to satisfy application queries.

A mitigating argument against this negative evaluation is that an Ada/SQL application programmer is free to "package" the Ada/SQL queries in a form similar to the abstract interface used by SAME applications. Although this would indeed be a good coding practice, it is not enforced by Ada/SQL. This would also not remove the two-pass compilation requirement.

5.2.6 Performance – Run Time

This negative evaluation of Ada/SQL is more cut-and-dry. The Ada/SQL approach prohibits the possibility of compile-time query optimization. Thus, applications requiring tight performance constraints on time-critical DBMS accesses would not be well served by Ada/SQL. On the other hand, the separate sourcing of SQL allows the vendor supplied SQL compiler full opportunities to optimize queries.

A possible mitigating argument against this negative Ada/SQL evaluation would be to note that in practice such time critical queries would be few, and that in such cases it is possible to re-write the Ada/SQL code performing the query to make use of a commercial embedded-SQL product. This would provide access to compile-time query optimization for the few queries that require it, while maintaining a pure Ada solution elsewhere. This does not, however, seem to be a satisfying counter-argument.

5.2.7 Ease of Use

This is a difficult criterion to assess, since it is granted that the SAME approach as it stands introduces considerable complexity for database application programming. On the other hand, public domain SAME automation tools are expected to be available to the Ada community by some time in August, 1989.

Nevertheless, *as currently implemented*, there is no question that Ada/SQL is both conceptually and practically easier to use than the SAME solution.

5.2.8 Support DB Design Methods

As mentioned earlier in this report, the SAME supports sound database application design principles by requiring early (design time) identification of required DBMS services. In contrast, Ada/SQL (and all other embedded SQL approaches) encourage an ad hoc approach to this part of application design; apparent design decisions can be made "on the fly" during the coding phase.

The elevation of database interface specification to a key design activity is a major improvement on the process of developing database application software in Ada, and meshes well with current software engineering design principles.

5.2.9 Effective Use of Ada

Both Ada/SQL and the SAME are satisfactory in this respect.

5.2.10 Effective Use of SQL

Again, separate sourcing of Ada and SQL has its advantages. The SQL standard is essentially an *intersection* standard, based upon a common set of SQL features provided by most SQL DBMS vendors. In practice this means particular implementations of SQL will provide additional services not defined in the SQL standard. Separate sourcing of SQL allows database applications to take advantage of these additional features.

The reader may question whether it is in fact a good idea to take advantage of vendor-specific services, as this has clear ramifications on application portability. It would seem that constraints, e.g. use only "standard" SQL, are best made on a project to project basis. The SAME, in any event, does not prevent the use of non-standard but potentially very useful vendor specific services, and in this regard is more flexible than Ada/SQL.

5.2.11 Well Defined Type Mapping

Both the SAME and Ada/SQL are constrained by the same requirement: data manipulated in the Ada programs must have a corresponding concrete representation in the underlying DBMS. Both the SAME and Ada/SQL provide the Ada application programmer with a view of database types as abstract types, and so both are satisfactory in this regard.

The "?" is appended to the Ada/SQL evaluation to indicate that Ada/SQL may not define the type mapping as clearly as in the SAME. First, note that the SAME support library defines supported base types as abstract data types; the application domain types are derived from these base types. Thus in the SAME there is a well defined, isolatable description of the Ada–SQL type mapping system. Further, the mechanics of extending the SAME to support different abstract base types (e.g., decimal) is well defined even if the required code to implement the base type is not.

It also must be noted that Ada/SQL does *not* support arbitrary Ada data types. Clearly, access types are not included, and neither are record types (variant or otherwise) as this would violate required (minimally) first normal form. Thus, although Ada/SQL and the SAME are constrained by the same type mapping requirements, in the author's opinion the SAME tackles the problem with greater uniformity.

5.2.12 Data Independence, Schema Evolution

This point has been made in conjunction with earlier discussions of the equivalence of the SAME application's abstract module interface to the notion of external schema. SAME application programs (as distinguished from the abstract and concrete modules, which in practice may be shared among several applications) are relatively data independent and impervious to information preserving schema modifications. This much is absolutely enforced by the SAME. In contrast, Ada/SQL does not require the use of abstract ("external") interfaces to SQL services, and must be considered weaker in this regard.

5.2.13 Data Interoperability to non-Ada Applications

One consequence of the Ada/SQL approach is that the database schema definition is derived from Ada source definitions of records and types. This is reasonable only in environments where Ada is the sole (or perhaps principle) language in use. In multi-lingual environments, especially those that employ database 4GLs, this Ada-orientation introduces problems, e.g., configuration management problems.

In the SAME, Ada is considered just another player in the database environment. Thus, single schema definitions may be shared, easing configuration management problems as well as making the database concrete column type representations available to all applications in arbitrary languages.

One mitigating argument which can be made is that the schema definition generated from the Ada/SQL data dictionary package can be shared by non-Ada applications. However, such access must be restricted to "read-only" in order to ensure Ada source compatibility with the database. This is a configuration management problem which arises since the Ada compilation system will not track dependencies between Ada source and the DBMS data dictionary. Further, recompilation of the Ada/SQL source, even if out of date with respect to the DBMS schema, will still succeed (in generating incorrect code). In contrast, recompilation of SAME SQL module code will detect inconsistencies with the DBMS data dictionary *at compile time.*

5.2.14 Ease of Validation

This criterion is more a conjecture than the other criteria, as is reflected in the "?" suffixes to both evaluations. The issue of validation arises by virtue of Ada compiler validation: can similar requirements be placed upon the Ada–SQL interface? The first point to note is that it is extremely unlikely that an underlying DBMS will be "validated", whatever that would mean. Nonetheless, it might be useful to validate the interface *to* the DBMS.

Again, separate sourcing is the key to the favorable (albeit tentative) endorsement of the SAME. The SAME interface is *thinner* than that of Ada/SQL precisely because it does not embed SQL;

thus there is no reason to validate that portion of the interface. Further, the additional tooling required of Ada/SQL to filter unnecessary column operations requires validation. This is so because the pre-compiler must perform full Ada overload resolution; it is, in fact, a compiler frontend.

The SAME support library would need validation; however, this should be straightforward since it defines for the most part the Ada-SQL type mapping, and this code is straightforward. Further interfaces requiring validation would be identified on a per application basis as new abstract and concrete modules are constructed. Note that the application use of domain types would not require validation, since they are derived from the SAME support library.

Finally, the validation of abstract and concrete modules could also be simplified by validating the SAME interface generator. The idea would be to prove the generated code valid by construction, and is indeed one of the salient benefits of the approach being taken to construct the SAME compiler.

6 Conclusions

In the author's opinion, the SAME approach to proving an interface from Ada to SQL should be pursued by STARS. Although the Ada/SQL approach has some noteworthy characteristics, the weight of evidence suggests that much greater effort will be required to transform this approach into a product quality interface than is the case for SAME. Besides the immediacy of possible product quality SAME implementations resulting from the disentanging of SQL from Ada (and thus leveraging existing product quality SQL compilers), the SAME also supports sound database application design principles, and seems more in the "spirit" of Ada and software engineering than other alternatives. Finally, the SAME provides concrete benefits not provided by any alternative, most notably robust error processing and safe treatment of SQL null values.

The SAME will require automation to achieve widespread acceptance; the interface programming task, although conceptually simple, is tedious. STARS should adopt the SEI SAME prototype for use in the STARS SEE. Becuase this prototype will be constructed using SSAGS, the SAME compiler should be fairly maintainable: there would be some flexibility in terms of definition of the SAME input language, should the syntax prove unacceptable to SAME users.

References

[BF88] Bill Brykczynski and F. Friedman. *Ada/SQL Binding Specifications.* Technical Report M-362, Institute for Defense Analysis, 1988.

[BFHH87] Bill Brykczynski, F. Friedman, F. Hilliar, and K. Hook. *Level 1 Ada/SQL Database Language Interface User's Guide.* Technical Report M-360, Institute for Defense Analysis, 1987.

[Boy87] Stowe Boyd. *SQL and Ada: The SQL Module Option.* Technical Report CA-8708-1701, Computer Associates, August 1987.

[Cle85] Erik K. Clemens. *Data Models and the ANSI/SPARC Architecture,* chapter 2, pages 66–114. S. Bing Yao (editor) Principles of Database Design, 1985.

[Dat87] C.J. Date. *A Guide to THE SQL STANDARD.* Addison-Wesley, 1987.

[EFGW87] Charles Engle, Robert Firth, Marc H. Graham, and William Wood. *Interfacing Ada and SQL.* Technical Report SEI-87-TR-48, Software Engineering Institute, December 1987.

[Gra88] Marc H. Graham. *Guidelines for the Use of the SAME.* Technical Report SEI-88-MR-9, Software Engineering Institute, October 1988.

[PKP*82] Teri Payton, S. Keller, J. Perkins, S. Rowan, and S. Mardinly. Ssags: a syntax and semantics analysis and generation system. In *Procedings of COMPSAC '82,* 1982.

[X3H86] ANSI Technical Committee X3H2. Database language – sql. 1986. X3.135-1986.

[X3H88] ANSI Technical Committee X3H2. Draft proposed american national standard embedding of sql statements into programming languages. 1988. X3.168-198x.